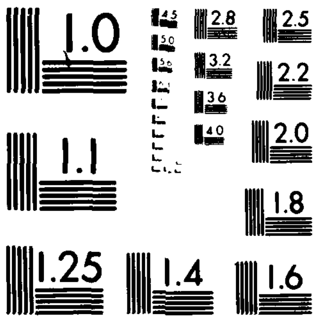
AD-A089 913 STANFORD UNIV CA DEPT OF COMPUTER SCIENCE F/G 9/2
READ-ONLY TRANSACTIONS IN A DISTRIBUTED DATABASE.(U)
APR 80 H GARCIA-MOLINA, G WIEDERHOLD N00039-80-G-0132
UNCLASSIFIED STAN-CS-80-797 NL

1 OF 1
AD A
089913

END
DATE
FILMED
11-80
DTIC



MICROCOPY RESOLUTION TEST CHART

NATIONAL BUREAU

12

Stanford Department of Computer Science
Report No. STAN-CS-80-797

April 1980

Princeton Department of Electrical Engineering
Technical Report No. 267

READ-ONLY TRANSACTIONS IN A DISTRIBUTED DATABASE

by

Hector Garcia-Molina
Princeton University

and

Gio Wiederhold
Stanford University

Research sponsored by

Advanced Research Projects Agency

COMPUTER SCIENCE DEPARTMENT
Stanford University

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

12 | 27

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER STAN-CS-80-797	2. GOVT ACCESSION NO. AD-A089913	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) 6 Read-Only Transactions in a Distributed Database.		9 5. TYPE OF REPORT & PERIOD COVERED technical rept., ~~April 1980~~
		14 6. PERFORMING ORG. REPORT NUMBER STAN-CS-80-797
7. AUTHOR(s) 10 Hector Garcia-Molina [redacted] Gio Wiederhold		15 8. CONTRACT OR GRANT NUMBER(s) [redacted] N00039-80-G-0132
9. PERFORMING ORGANIZATION NAME AND ADDRESS Stanford University Department of Computer Science Stanford, California 94305 USA		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Defense Advanced Research Projects Agency Information Processing Techniques Office 1400 Wilson Avenue, Arlington, Virginia 22209		11 12. REPORT DATE Apr[redacted]80 — 13. NO. OF PAGES 23
14. MONITORING AGENCY NAME & ADDRESS (if diff. from Controlling Office) Mr. Philip Surra, Resident Representative Office of Naval Research, Durand 165 Stanford University		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

(see other side)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE 094120 [illegible]

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

19. KEY WORDS (Continued)

20 ABSTRACT (Continued)

A read-only transaction or query is a transaction which does not modify any data. Read-only transactions could be processed with general transaction processing algorithms, but in many cases it is more efficient to process read-only transactions with special algorithms which take advantage of the knowledge that the transaction only reads. This paper defines the various consistency and currency requirements that read-only transactions may have. The processing of the different classes of read-only transactions in a distributed database is discussed. The concept of *R* insularity is introduced to characterize both the read-only and update algorithms. Several simple update and read-only transaction processing algorithms are presented to illustrate how the query requirements and the update algorithms affect the read-only transaction processing algorithms.

DD FORM 1 JAN 73 1473 (BACK)

EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Stanford Department of Computer Science
Report No. STAN-CS-80-797

April 1980

Princeton Department of Electrical Engineering
Technical Report No. 267

READ-ONLY TRANSACTIONS IN A DISTRIBUTED DATABASE

by

Hector Garcia-Molina
Princeton University

and

Gio Wiederhold
Stanford University

ABSTRACT

A read-only transaction or query is a transaction which does not modify any data. Read-only transactions could be processed with general transaction processing algorithms, but in many cases it is more efficient to process read-only transactions with special algorithms which take advantage of the knowledge that the transaction only reads. This paper defines the various consistency and currency requirements that read-only transactions may have. The processing of the different classes of read-only transactions in a distributed database is discussed. The concept of *R* insularity is introduced to characterize both the read-only and update algorithms. Several simple update and read-only transaction processing algorithms are presented to illustrate how the query requirements and the update algorithms affect the read-only transaction processing algorithms.

This research was supported by the Advanced Research Projects Agency of the Department of Defense under Contract N00039-80-C-0132. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of Stanford University, or any agency of the U. S. Government.

READ-ONLY TRANSACTIONS IN A DISTRIBUTED DATABASE

Hector Garcia-Molina

Department of Electrical Engineering and Computer Science
Princeton University
Princeton, N.J. 08544

Gio Wiederhold

Department of Computer Science
Stanford University
Stanford, CA. 94305

1. INTRODUCTION.

Distributed databases have become a very active research area. Within this research area, the transaction processing problem has been getting considerable attention. A transaction is (informally) a user application request or program that has been submitted to the distributed database. The distributed database system must efficiently process these transactions in a way which does not violate the data integrity or consistency constraints [Eswa76].

For example, suppose that data on a person's salary is duplicated at two nodes (sites) in the system. Since the two data items represent the same person's salary, the two items should have the same value. However, if two transactions attempt to modify the salary concurrently, it is possible for the two data items to end up with different values. A transaction processing algorithm, or concurrency control mechanism, is needed to prevent these types of problems. A large number of such algorithms for distributed databases have been proposed [Alsb76, Elli77, Gray77, Thom79].

A read-only transaction or *query* is a transaction which does not modify the distributed database. For example, a transaction to check the balance of a given checking account in a bank is a query. Since queries are still transactions, they can be processed using the algorithms for arbitrary transactions. However, it is also possible to use special processing algorithms for queries in order to improve efficiency. With this approach, the specialized algorithm can take advantage of the knowledge that no data will be modified by the transaction. In this paper, we will explore these ideas on queries, we will study the requirements for these special transactions, and we will analyze the types of algorithms needed for processing them. The emphasis of this paper will be on discussing the issues and available design choices instead of on the presentation of a particular approach.

The idea of using the "type" of a transaction in order to improve efficiency is not new [Bern78]. However, this has been a somewhat controversial idea because it is not clear if it is possible to know beforehand all of the transaction types that will run on a distributed database system. Nevertheless, we believe that the transaction "type" idea does have great potential in the special case where there are only two types: read-only transactions and update (i.e. not read-only) transactions. The reasons for this are that (1) we believe that there will be a high percentage of read-only transactions in most systems, and (2) we believe that it will be fairly easy to distinguish a read-only transaction from an update transaction.

In this paper we will avoid two important issues of transaction processing: directory management and transaction optimization. That is, we will not consider how a transaction locates the data it needs, nor will we consider how a transaction decides where and in what order to read the data it needs. We will assume that by the time that a user transaction is given to the transaction processing algorithm, the user transaction has been translated into a series of actions which reference particular data values located at particular nodes.

In order to simplify the presentation, we will make another important assumption about the distributed database system. We assume that no failures occur in the system. This is a reasonable assumption because crash recovery for queries is straightforward. Since a query does not modify the data, there is no recovery problem with an unfinished read-only transaction.

In the following section we will present the basic concepts that are needed for this paper. Then in section 3, we will study read-only transactions and the various user requirements that they may have. In section 4 we will discuss how the different classes of queries can be processed efficiently. Finally, some conclusions from this study are presented in section 5.

2. BASIC CONCEPTS.

In this section, we will review some of the basic concepts that will be useful in our study of queries. Many of the concepts in this section are from [Eswa76], adapted to a distributed database system. (The concept of consistency for a distributed database is the same as for a centralized database. However, the concurrency control mechanisms for maintaining consistency in a distributed database differ significantly from those in a centralized database.)

A distributed database is a collection of named *items.* Each item has a *name* and between 1 and *n* *values* associated with it, where *n* is the number of nodes in the system. Each value for a given item is stored at a different node of the system. In addition, each item *i* has associated with it a set $S(i)$. Set $S(i)$ is the set of nodes which have a value for item *i* stored in them. We assume that all sets $S(i)$ are not empty, and at most have all *n* nodes in them. We use the notation $d[i,x]$ to represent the value of the item named *i* at node *x*. For nodes *y* not in $S(i)$, $d[i,y]$ is undefined. The values for a given item *i* at different nodes should be the same (i.e., $d[i,x]$ should equal $d[i,z]$ for all nodes x,z in $S(i)$). However, due to the updating activity, the values may be temporarily different.

Figure 1 shows a sample distributed database. There are four items in this distributed database: item "deposits" represents the total deposits made to a certain bank account, item "withdrawals" represents the total withdrawals from the same account, item "balance" represents the balance of the account, while item "name" represents the name of the person who owns the account. As can be seen in figure 1, the balance of the account is reported to be 60 dollars at nodes 1 and 2. That is, d[balance,1] = d[balance,2] = 60. Since node 3 does not have a value for item "balance", d[balance,3] is undefined.

In addition to items and values, a distributed database has a collection of *consistency constraints.* A consistency constraint is a predicate defined on the database which describes the relationships that must hold among the items and their values. For example, the distributed database of figure 1 my have the consistency constraint "deposits – withdrawals = balance". In addition to this type of constraint, distributed databases always have the constraint that the values of an item should be equal. We call this constraint the *implicit* constraint simply to differentiate it from the other user dependent constraints.

We say that a distributed database is *consistent* (or is in a consistent state) if all the consistency constraints are satisfied by the data values.

We can use the $d[i,x]$ notation to express the consistency constraints. For example, the constraint "deposits-withdrawals = balance" becomes "d[deposits,1] – d[withdrawals,1] = d[balance,1]", and "d[deposits,2] – d[withdrawals,2] = d[balance,2]". Notice that there is no constraint at node 3 because two of the values involved are not defined there. The implicit constraint for the example becomes "d[deposits,1] = d[deposits,2]", "d[withdrawals,1] = d[withdrawals,2] = d[withdrawals,3]", and "d[balance,1] = d[balance,2]".

Operations on the data are grouped into transactions. Each transaction **T** is a sequence of actions which preserve consistency. One way of representing an action is by a triplet $(T, a, d[i,x])$ where T is the name of the transaction which performs the action, a is the action type (e.g., read, write), and $d[i, x]$ is the data value which is referenced by the action

($d[i,x]$ must be defined). Notice the difference between T, the name of the transaction, and **T**, the transaction itself. All transactions must preserve consistency. That is, if a transaction is run on a consistent distributed database and without interference from other transactions, then the transaction should leave the distributed database in a consistent state.

Figure 2 shows a transaction T_1 that registers a 5 dollar deposit into the account of figure 1. The transaction is presented in an informal notation which describes the actual operations performed. The formal notation for T_1 as a sequence of actions in also shown in figure 2. Notice that the formal representation of T_1 does not specify that the value that is written into d[balance,1] is 5 dollars more than its old value, nor that the values written into d[balance,1] and d[balance,2] are the same. However, as long as we know that T_1 preserves consistency, the information given in figure 2 is sufficient to study the potential conflicts of T_1 with other transactions.

A schedule **S** of a set of transactions $T_1, T_2, \ldots, T_m$ represents a particular order in which the actions of the transactions were performed in the system. A schedule is also a sequence of actions. The actions of schedule **S** are simply the actions of the transactions $T_1, T_2, \ldots, T_m$ interleaved in a way which preserves the relative order of the original actions. That is, if $(T_j, a_1, d[i_1,x_1])$ preceded $(T_j, a_2, d[i_2,x_2])$ in T_j (for $1 \leqslant j \leqslant m$), then $(T_j, a_1, d[i_1,x_1])$ must precede $(T_j, a_2, d[i_2,x_2])$ in **S**. Figure 3 shows a schedule for transactions T_1 and T_2, where transaction T_1 is given in figure 2, and transaction T_2 is a similar transaction which withdraws 10 dollars from the same account. The informal notation for **S** is also given.

Some schedules have undesirable effects. For example, if the schedule named S in figure 3 is run on a consistent distributed database, then the distributed database will be left in an inconsistent state. (Try it). Hence, it is important to discover which schedules are "good" and which have undesirable effects. One set of "good" schedules are those schedules which are serial or which are equivalent to serial. Serial schedules are ones where transactions are performed one at a time, and are clearly "good". A schedule which is equivalent to a serial schedule is one which has the same effect on the data as some serial schedule. We will call this set of "good" schedules *consistent schedules.* The following definitions and theorem [from Eswa76] state these concepts more precisely:

Definition 1. Let T be the set of all transaction names in a schedule named S, and let V be the set of all defined item values $d[i,x]$ in the distributed database. The dependency relation induced by schedule S,$dep(S)$, is a ternary relation on $T \times V \times T$ defined by $(T_1, v, T_2) \in dep(S)$ if and only if for some $i < j$

$$\mathbf{S} = (\ldots, (T_1, a_i, v), \ldots, (T_2, a_j, v), \ldots)$$

and a_i or a_j is a write type action and there is no k such that

$i < k < j$ and
$v_k = v$ and
$a_k =$ write.

The last part of the definition specifies that there should be no write action on item value v between the (T_1,a_i,v) action and the (T_2,a_j,v) action. (Notice that item values $d[i,x]$ play the role of the entities of [Eswa76]).

Definition 2. Let S_1 and S_2 be two schedules for transactions $T_1, T_2, \ldots, T_m$. S_1 is equivalent to S_2 *iff* $dep(S_1) = dep(S_2)$.

Definition 3. The binary relation "<" on the set of transactions of a schedule **S** is defined by: $T_1 < T_2$ *iff* $(T_1, v, T_2) \in dep(S)$ for some item value v.

Theorem 1. A schedule S is consistent (i.e., equivalent to a serial schedule) iff the binary relation < for S is an acyclic relation.

To illustrate the definition of consistent schedule, consider once more the schedule S of figure 3. Here, $(T_1, d[\text{balance},1], T_2) \in dep(S)$ because of the actions in lines 4 and 6.

Similarly, $(T_2, d[\text{balance},2], T_1) \in dep(S)$ because of the actions in lines 10 and 11. Therefore, $T_1 < T_2$ and $T_2 < T_1$ and the $<$ relation is not acyclic. So, S is inconsistent.

As we will see later on in this paper, there are some schedules which some users might consider "good" which are not "consistent" by the given definitions. However, all consistent schedules are "good" in the sense that they do not cause any undesirable effects or violate the distributed database consistency. Furthermore, it is relatively easy (with the theorem) to check if a schedule is consistent, so most researchers have chosen consistent schedules as their definition of "good" schedules. In this paper, we will also use this definition.

A *transaction processing algorithm* should process transactions in such a way that the resulting schedule is consistent. A few of these algorithms will be described in section 4.1.

3. READ-ONLY TRANSACTIONS.

A read-only transaction or query is a transaction without any write actions. Such a transaction simply reads data from the distributed database and presents the values obtained to a user of the system. The user cannot then make an update to the distributed database based on the data obtained from the query. If the user wishes to submit such an update, the update transaction must first read the data again to check if the data has changed.

We believe that in many distributed database applications queries represent a very significant proportion of the total transactions submitted. Hence it is important to understand what classes of queries exist as well as how they can be processed efficiently.

3.1 Requirements for Queries.

Users who issue queries may have varying requirements for the transactions. These user requirements can be divided into two independent classes: the consistency and the currency requirements.

The consistency requirements specify the "degree" of consistency needed by the read-only transaction. ([Gray76] discusses degrees of consistency for arbitrary transactions; here we will define different but related degrees for queries). A query may have strong, weak or no consistency requirements. If a query Q_1 has no consistency requirement, then Q_1 may read data which is inconsistent. If a query Q_2 requires *weak consistency*, then Q_2 must obtain a consistent view of the data. This means that all the consistency constraints that can be fully evaluated with the data read by Q_2 must be true. If a query Q_3 requires *strong consistency*, then the schedule of all the update transactions (i.e. not read-only) together with all other strong consistency queries must be consistent. Notice that Q_3 also obtains a consistent view of the data. (Since a consistent schedule is equivalent to some serial schedule, all transactions in the schedule read consistent data). Thus, all queries with the strong consistency requirement also satisfy the weak consistency requirement. In section 4.1.2 we will see that queries with the weak requirement do not necessarily satisfy the strong requirement. The example in that section will also clarify the difference between the weak and strong requirements.

The currency requirement of a query specifies what update transactions should be reflected by the data read. There are several ways in which the currency requirement can be stated; here we will discuss three common ways.

(1) *t-vintage* requirement. A query Q_4 can require data as it existed at a given (previous) time t. This means that the data read by Q_4 must reflect the modifications of all update transactions "committed" in the distributed system before time t, and must not reflect modification due to any update transactions "committed" after time t. Intuitively, the commit time of an update transaction T is the time when any data modification produced by T first becomes available anywhere in the system. In many cases, the time of the first write action of T at any node would be the commit time of T. However, if the data written by T is protected by locks or any such mechanism which makes the new value written inaccessible, then the commit point is not reached until the new values actually become accessible to other transactions. (Here, as in the rest of the paper, we use the concept of

time in an intuitive fashion. These ideas could be formalized using the concepts in [Lamp78]). We will call transactions like R_4 t-vintage queries.

(2) *t-bound* requirement. A query Q_5 can request that the data it reads at least reflect all update transactions committed before time t. The data may or may not reflect any transactions committed after time t. We will call transactions like Q_5 t-bound queries. A special case occurs when t is the current time (i.e., the time when Q_5 is submitted). In this case, Q_5 requires the latest or most up-to-date data available. Such transactions will be called *latest-bound* queries.

(3) No currency requirement. A query Q_6 can have no currency requirements. This means that the data read by Q_6 can reflect any set of update transactions.

3.2 An Example.

Let us illustrate the consistency and currency requirements with a simple example. Consider once again the distributed database of figure 1 but for simplicity, let us assume that node 3 does not exist. Suppose that there are 3 update transactions T_1, T_2 and T_3. Transactions T_1 (shown in figure 2) is submitted at time t_1 and records a 5 dollar deposit to the account. Transaction T_1 reads the total deposits and balance at node 1, updates these values at node 1, and sends a message M_1 to node 2 indicating what these changes are. The message in this case would be "d[deposits,2] $\leftarrow$ 105; d[balance,2] $\leftarrow$ 65" because the deposits and balance read at node 1 were 100 and 60 respectively. Next, transaction T_2 is submitted at time t_2 ($t_2 > t_1$) and records a 10 dollar withdrawal. Transaction T_2 reads the total withdrawals and balance at node 1 (which now reflect T_1), updates these values at node 1, and sends a message M_2 to node 2. Since the withdrawals and balance at node 1 were 40 and 65 respectively, M_2 is "d[withdrawals,2] $\leftarrow$ 50; d[balance,2] $\leftarrow$ 55". Next, transaction T_3 is submitted at time t_3 ($t_3 > t_2$) and records a 100 dollar deposit. After reading and updating node 1, T_3 sends the message M_3 to node 2: "d[deposits,2] $\leftarrow$ 205; d[balance,2] $\leftarrow$ 155".

Figure 4 shows the situation at this point. Let us assume that messages M_1, M_2, and M_3 can arrive in any order at node 2. Let us define M_i (d) to be the resulting item values at node 2 after message M_i is processed there, given that d are the previous item values. For example, if d_0 are the initial values at node 2 (see figure 1), then $M_1(d_0)$ are the values d[deposits,2] = 105, d[withdrawals,2] = 40 and d[balance,2] = 65. Notice that if the messages are processed out of order at node 2, then consistency is violated. For example, $M_1(M_2(d_0))$ is inconsistent because the resulting deposits minus the resulting withdrawals is not equal to the resulting balance.

Next, a query Q arrives at node 2 at time t_4 ($t_4 > t_3 > t_2 > t_1$). Query Q requests the values of all three items at node 2. If Q is a weak consistency query (i.e., a query with a weak consistency requirement), then Q can only be allowed to read d_0 (the initial values), $M_1(d_0)$, $M_2(M_1(d_0))$ or $M_3(M_2(M_1(d_0)))$. If Q is a strong consistency query, then it can only read those same consistent values. However, if Q has no consistency requirement, it can read anything: d_0, $M_1(d_0)$, $M_1(M_2(d_0))$, $M_2(M_1(d_0))$, and so on.

If query Q is t-vintage where $t_2 < t < t_3$, then it should read data that reflects only those transactions submitted before time t, that is T_1 and T_2. Hence, in this case, Q can read $M_2(M_1(d_0))$ or $M_1(M_2(d_0))$ only. If in addition to being t-vintage, Q is a weak consistency (or strong consistency) query, then only the values $M_2(M_1(d_0))$ would be acceptable. If Q is a t-bound query with $t_2 < t < t_3$, then Q can read $M_2(M_1(d_0))$, $M_1(M_2(d_0))$, $M_3(M_2(M_1(d_0)))$, $M_2(M_1(M_3(d_0)))$, and any other data which reflects M_1, M_2, M_3 in any order. However, neither $M_3(M_2(d_0))$ nor $M_1(M_3(d_0))$ would be acceptable to this t-bound query. If in addition to being t-bound, Q is a weak consistency (or strong consistency) query, then only the values $M_2(M_1(d_0))$ or $M_3(M_2(M_1(d_0)))$ would be acceptable to Q. If Q is a latest bound (i.e., t_4-bound) query, then any data which reflects M_1, M_2, M_3 in any order is acceptable, but if Q is also weak (or strong) consistent, then only $M_3(M_2(M_1(d_0)))$ is acceptable. Finally, if Q has no currency requirement, then it can read anything: d_0, $M_1(d_0)$,$M_2(d_0)$, $M_2(M_1(d_0))$,$M_1(M_2(d_0))$, etc.

In this example we have not described how the query Q could go about reading the data it needs. (We did not even consider reading data at node 1.) The example simply discussed what data would be acceptable to Q depending on its user defined consistency and currency requirements. The algorithms for processing the different types of queries will be discussed in section 4. The example did not illustrate the difference between weak consistency and strong consistency queries. An example which illustrates this difference will be given in section 4.1.2.

3.3 Why Different Read-only Transaction Types are Needed.

In this section we will discuss why the different query classes are needed. For instance, it might seem that a query with no currency or consistency requirements is not very useful because, as the previous example illustrated, it may produce data which " does not make much sense". On the other hand, queries with no requirements, which we will call *free* queries, are extremely simple and efficient to process. All that has to be done to process free queries is to find the nearest copy of the requested data and read it, without worrying about the consistency of the data or how old the data is. So, in many applications, users may be willing to sacrifice consistency and currency for efficiency. Furthermore, in a well designed distributed database system, free queries should produce results which are not too old. For example, a warehousemanager might want a rough idea of where the inventory stands. The manager does not really care if the data obtained is 15 or 30 minutes old. The manager might not mind that the total number of parts reported does not exactly match the sum of the itemized entries in the report obtained (which may be a consistency violation). As another example, consider a free query which computes an average salary for a large set of employees. The result might not be accurate if some of the salaries are being updated during the long period that the averaging query is running. But the user might decide that such occasional conflicts will not alter the average significantly. Furthermore, not running the averaging query as a free query will produce long delays in other transactions that access the salary data.

Another case where free queries are valuable is in one item queries. A query that only reads one item value will always give a consistent view of the data. (The last update transaction that wrote a value into the item must have made sure that the written value satisfied any consistency constraints dealing exclusively with the item.) Therefore, the simple algorithm for processing free queries can be used for one item queries and the result will always be consistent. In many systems, one item queries are common and considerable effort can be saved if we use an efficient method like the free query mechanism for performing these queries.

Clearly, not all queries in a system can be free. In some queries, a consistent view of the data is required. For instance, the checking account monthly statement that is sent to a bank customer must be consistent. (E.g., the sum of the cashed checks should equal the total debits entry). Usually, the query to print the monthly statement does not have any currency requirements because any checks missed this month will simply be reported next month. (If the checking account yields an interest, then there may be some currency requirements).

For an example of a t-vintage query, consider the case of a tax auditor examining the computerized records of a company. The auditor may be interested in looking at the database as it existed December 31, 1960 at 12:00 midnight. Queries of December 31, 1960, 12:00 midnight - vintage will provide the auditor with the desired results. Such queries would probably be weak (or strong) consistent too. But in some cases, like a one item transaction, no consistency requirement may be necessary.

In some situations it may be necessary to obtain the latest information that is available in the system. For example, a general who has to decide whether to fire or not a missile at an incoming airplane will use a latest-bound query to obtain the latest information on the airplane's position and speed.

To see when a t-bound query with t different from the current time could be used, consider the case of a distributed database node which becomes isolated from the rest of the system at time t_1 because of a communications failure. (True, we have assumed that no failures occur, but let us make this small exception). In this case, any latest-bound queries submitted after time t_1 at the isolated node will be delayed until the failure is repaired. (While the node is isolated, it cannot tell if other update transactions are being processed by the rest of the system). Instead of waiting, users may prefer to submit t_1-bound queries to at least obtain the latest data available before the crash.

4. PROCESSING QUERIES.

It is now time to discuss how queries can be processed. The approach we take in this paper is that queries are processed with a special algorithm which is different from the update transaction processing algorithm. This way, queries can be processed more efficiently because the algorithm for processing them can take advantage of the fact that no data will be modified by these transactions.

In order to use a special query algorithm, we must assume that it is possible to a priori distinguish queries from the other transactions. This is a reasonable assumption because the transaction programmer can usually tell if the transaction has no intention of updating the database. Hence, transactions can be marked as read-only from their inception. We will also assume that the consistency and currency requirements for each query are given when the query is initiated. If the requirements are unknown, then some standard default requirements could be made.

Although it is difficult to evaluate the efficiency of query processing algorithms, it is possible to define one evaluation criterion which is very useful. Suppose that a query Q is submitted at a certain node x (by a user located at node x). We say that query Q is *insular* if all the items referenced by Q have values at the node where Q was submitted. Depending on the consistency and currency requirements of insular query Q, the query processing algorithm may or may not be able to process Q locally at node x, without the need to communicate or synchronize with other nodes. If an algorithm can process an insular query Q with requirement R at the node where Q was submitted, we say that this is a R *insular* algorithm. For example, an algorithm which can process weak consistency insular queries at the submission node of the query is a weak consistency insular algorithm. We expect a R insular algorithm to be a considerably superior to one which is not R insular, at least as far as R queries are concerned. Thus, R insularity is an important characteristic of an algorithm which can be used for comparing and evaluating query processing algorithms.

At this point it would be nice if we could simply present one query algorithm which is R insular for all possible requirements R. Unfortunately this is not possible because the query algorithm is closely related to the algorithm used for processing update transactions. Thus, for each possible update algorithm, we have to design a different query algorithm. Furthermore, with some update algorithms it is not possible to have query algorithms with all the desirable properties. For example, with certain update algorithms it is impossible to process weak or strong consistency insular queries at their originating node. That is, for those update algorithms there are no weak or strong consistency insular algorithms.

This means that we can also use the R insular property to characterize update algorithms. We say that an update transaction processing algorithm is R *insular* if there exists an R insular query algorithm that can be used with that update algorithm. We consider R insularity an important feature of update algorithms because efficient query processing hinges on those features.

In the following section we will present some examples to illustrate some of the concepts we have defined so far. Then in section 4.2, we will consider the issues involved in processing non-insular queries. Queries with t-vintage and t-bound requirements will not be covered until section 4.3.

4.1 Examples of R Insular Update Algorithms.

In this section we will present four update algorithms with various interesting R insularity properties. Recall that an update algorithm is R insular if there exists a query algorithm capable of locally processing insular queries with requirement R. Since we are interested in insular queries, let us assume in this section that the data is *completely replicated* at all nodes of the system. That is, assume that all items i have a value $d[i,x]$ at every node x of the system. This will make all queries insular and will simplify our examples. After the examples, we return to the general case in order to discuss non-insular queries.

Before starting with the examples, let us point out that the update algorithms we will present are extremely simple, and possibly not very efficient. The purpose of the examples is to illustrate how queries can be processed and not how to design efficient update algorithms. But let us also point out that the algorithms we will show are just simplified versions of some popular (and more efficient) update algorithms, where both the simple and the original algorithms have the same features with respect to queries.

4.1.1. An Example: The Complete Centralization Algorithm (CCA).

The first sample update processing algorithm is a strong consistency insular one. The name of the algorithm is the Complete Centralization Algorithm (CCA) (inspired by the primary site algorithm of [Alsb76]).

In the CCA, a node of the distributed database is selected as the "central" site. This central node processes all update transactions one at a time, and sends messages to all other nodes giving them the new values. We now give a brief description of the CCA. Figure 5 depicts the steps of the CCA in a 4 node system.

(1) Update transaction T arrives at node x from a user. (Recall that we assume that all data is replicated at every node in the system)

(2) Node x forwards transaction T to the central node.

(3) When the central node receives T, it places it in a queue. Update transaction T waits in the queue until its turn to be executed comes up.

(4) When T's turn comes, it is executed at the central node. (At this point, all previous transactions have completed at all nodes). The item values requested by T are read from the database at the central node, any necessary computations are carried out, and the new values are stored in the local database.

(5) "Perform update T" messages are sent out by the central node to all other nodes giving them the new values that must be stored at each site.

(6) Each node that receives a "perform update T" message stores the new values produced by T into the database. Then an acknowledgement message is sent back to the central node.

(7) When the central node receives acknowledgements for the "perform update T" messages from all the nodes in the system, then it knows that T has completed everywhere. Thus, the central node gets the next update transaction that is waiting in its queue and processes it. (See step 4.) (End of CCA.)

The CCA processes update transactions one at a time, without interleaving with other update transactions. Therefore, any schedule of update transactions produced by the CCA is serial and hence consistent.

Now suppose that a query Q, which desires a consistent view of the database, arrives at a node x. Since all items have local values, query Q is insular. Notice that between update transactions, the distributed database is consistent. Hence, if Q reads its data at node x "between" update transactions, it will get a consistent view. In other words, node x can process Q by waiting until the current update transaction (if any) completes at node x, and then by delaying any actions of the following update transaction until Q completes. (The following transaction can be delayed by delaying the acknowledgement for the previous one). What we have just described is a local processing algorithm for queries. Let us call this algorithm the atomic algorithm (AA) because queries are performed as if they were atomic operations.

The AA gives a consistent view of the data and is therefore a weak consistency insular query algorithm. As a matter of fact, the AA is also a strong consistency insular algorithm. To show this, we must show that any schedule produced by the CCA and the AA is consistent (although this may be obvious to some readers).

Instead of immediately showing that the AA is strong consistency insular, let us briefly describe another significantly more efficient query processing algorithm which can also be used for processing strong (and weak) consistency insular queries. After presenting the second algorithm, we will show that both algorithms have the strong consistency insular characteristic.

The local locking algorithm (LLA) uses local locks to guarantee that the local queries do not conflict with the update transactions. A local lock is assigned to every value $d[i,x]$ at node x (for all nodes x). Before any transaction (query or update) can reference (i.e. read or write) an item value, it must request the lock for that value. Once obtained, the lock gives that transaction exclusive access to the value. When the transaction completes at node x (but possibly not at other nodes), all local locks held by the transaction are released. Other waiting transactions may then be given the newly released locks. Notice that the local lock manager (which is just the LLA) does not need to communicate with other nodes in order to decide whether to grant or release a lock. (In the CCA, a node can easily tell which is the last action of an update transaction at a node. It is the last write of the "perform update" message. In other update algorithms, it is also possible to identify the last action of a transaction at a node.)

The LLA gives queries a consistent view of the data because whenever a query Q sees the effects of an update transaction T_1 then Q sees the complete effects of T_1, and also, Q sees the complete effects of any update transaction T_2 which preceded T_1. In other words, the LLA guarantees than in any schedule S, the following conditions hold:

Condition 1: $(T_1, d[a,x], Q) \in dep(S)$ implies that $(Q, d[b,x], T_1) \notin dep(S)$, for any update transaction T_1, query Q, items a, b, node x.

Condition 2: $(T_1, d[a,x], Q) \in dep(S)$ and action $(T_2, w, d[b,x])$ precedes action $(T_1, w, d[a,x])$ at node x implies that $(Q, d[b,x], T_2) \notin dep(S)$, for all update transactions T_1, T_2, query Q, items a,b, node x.

It is simple to check that the LLA does indeed force all schedules to satisfy these conditions. For condition 1, $(T_1, d[a,x], Q) \in dep(S)$ means that S must be of the form

$$(\ldots, (T_1, w, d[a,x]), \ldots, (Q, r, d[a,x]), \ldots).$$

This implies that T_1 must complete at x before Q does (or else Q would not get the $d[a,x]$ lock). If $(Q, d[b,x], T_1) \in dep(S)$, then we get the contradiction that Q completes at node x before T_1 does. Therefore, $(Q, d[b,x], T_1)$ must not be in $dep(S)$. The check for condition 2 is similar and is left as an exercise.

We can no show that the schedules produced by the CCA and LLA are consistent, and thus, strong consistency queries can be processed locally. As a matter of fact, any query algorithm that satisfies conditions 1 and 2 has this property. For example, the AA which processes queries as if they were atomic operations also satisfies these conditions an is hence also strong consistency insular.

Before presenting the following theorem, let us define a binary relation which will be useful in the theorem's proof. Let binary relation "<<" be defined on the set of query and update transactions of a schedule S as follows: $T_1 << T_2$ iff all actions of T_1 precede all actions of T_2 in S. Notice that "<<" is a transitive and irreflexive relation. That is, $T_1 << T_2$ and $T_2 << T_3$ implies $T_1 << T_3$, and $T_1 << T_1$ is false, for all transactions T_1, T_2 and T_3.

Theorem 2: All schedules produced by the CCA and an insular query processing algorithm which satisfies conditions 1 and 2 are consistent.

Proof: Suppose that there is a cycle in the "<" relation for S. This cycle can have update transactions or queries. Choose any update transaction (there must be one) and call it T_1. Call the other *update* transactions along the cycle $T_2, T_3, T_4, \ldots, T_m$. If a given update transaction appears more than once, it simply gets two names. In particular, $T_1 = T_n$. Now take any pair of update transactions T_i, T_{i+1}. There are two cases: (1) $T_i < T_{i+1}$. This implies that $T_i << T_{i+1}$ because the CCA is serial. (2) $T_i < Q < T_{i+1}$ for some query Q. (There can only be one query between T_i and T_{i+1} because queries cannot have dependencies). Here, $T_i \neq T_{i+1}$ because of condition 1 of the query algorithm. By condition 2, there is an action of T_i which precedes T_{i+1}, so $T_i << T_{i+1}$ because the CCA is serial. Since for every pair T_i, T_{i+1} we have $T_i << T_{i+1}$, then we must have $T_1 << T_2 << \cdots << T_n$. This, in turn, implies that $T_1 << T_n$ (since "<<" is transitive) which is impossible because $T_1 = T_n$. Therefore, all schedules must be consistent.

The LLA is an efficient algorithm for processing strong (and weak) consistency queries. In this section we illustrated how it could be used in conjunction with the CCA. However, the LLA is not limited to the CCA; the LLA can be used with a large variety of update algorithms. For example, in section 4.1.2 we will see how the LLA can be used with one more update algorithm.

Before ending this section, let us point out that deadlocks can occur with the LLA. In any situation where processes (i.e. transactions) compete for a finite set of resources (i.e. local locks), deadlocks may arise. Fortunately, all possible deadlocks are local ones, and can be easily detected by special processes running at each node. (A deadlock is local when all the resources involved in a cycle of waiting processes are resources located at a single node.) Here we assume that such deadlock detection processes do exist, and that they can successfully break deadlocks. Furthermore, we assume that the local lock manager at each node is fair and does not allow a transaction to wait indefinitely for a lock.

4.1.2 An Example: The Wait-for List Centralized Algorithm (WLCA).

In this section we will present an update algorithm which allows local queries to obtain a consistent view of the data, but where the schedule of all transactions may be inconsistent. The algorithm is the Wait-for List Centralized Algorithm (WLCA) (inspired by the centralized locking algorithm with wait-for lists of [Garc79]).

The WLCA is similar to the CCA in that a central node executes all update transactions. However, the central node does not wait for acknowledgements for the "perform update" messages before starting the next transaction. In order to prevent "perform update" messages from being executed out of order, a "wait-for" list is appended to each message.

The central node computes a wait for list, WFL(T), for each transaction T it processes. List WFL(T) contains the names of all transactions whose "perform update" messages must be processed before the message of T in order to avoid conflicts. (The outline of the WLCA which follows shows how the central node can compute these lists). A copy of wait-for list WFL(T) is appended to each "perform update" message of T, and nodes will only process these messages if they have processed all transactions in WFL(T) first.

We now give an outline of the WLCA. Array LAST is stored at the central node, and LAST(i) is the name of the last transaction which referenced (read or write) item i. A list, DONE(x), is kept at each node x. This list contains the names of transactions whose "perform update" messages have been processed at node x.

(1) An update transaction arrives at node x from a user. Node x assigns a name T to the transaction. (The name could be, for example, a node identification number, followed by the user name and a sequence number). Recall that we are assuming that all data is replicated at all nodes in the system).

(2) Node x forwards transaction T to the central node.

(3) When the central node receives T, it places it in a queue. Update transaction T waits in the queue until its turn to be executed comes up.

(4) When T's turn comes up, it is executed at the central node. (At this point, all previous update transactions have completed at the central node, but may still be active at other nodes). The item values requested by T are read from the database at the central node, any necessary computations are carried out, and the new values are stored in the local database.

(5) For each item i referenced by T, the central node adds LAST(i) to WFL(T) and then makes LAST(i) equal to the name T.

(6) "Perform update T" messages, which include copies of WFL(T) and T's name, are sent out by the central node to all other nodes, giving them the new values that must be stored at each site. The central node is done with T, and goes on to process the next update transaction that is waiting in its queue. (See step 4).

(7) When a node x receives a "perform update T" message, it looks at WFL(T). If WFL(T) is contained in DONE(x), then the message is processed (i.e., the values produced by T are stored locally) and the name T added to DONE(x). Otherwise, the message is saved until the missing messages have been processed. No acknowledgements are sent to the central node.

In order to describe an important property of the WLCA, let us define a binary relation "$<<_y$" on the set of all transactions of a schedule S, as follows: $T_i <<_y T_j$ if and only if all actions of transaction T_i precede all actions of transaction T_j at node y in schedule S. Notice that "$<<_y$", like "$<<$", is transitive and irreflexive.

The important property of the WLCA is that if update transactions T_1 and T_2 reference a common item i, then either $T_1 <<_y T_2$ at all nodes y, or $T_2 <<_y T_1$ at all nodes y. To see why this is true, suppose that T_1 is processed first at the central node, and say that transactions R_1, $R_2, \ldots, R_m$ also reference item i and are processed after T_1 but before T_2. Then, T_1 will be in WFL(R_1), R_1 will be in WFL(R_2) , . . . and R_m will be in WFL(T_2). As we can see in step 7 of the WLCA, if T_1 is in WFL(R_1), then $T_1 <<_y R_1$ at all nodes y. Similarly, $R_1 <<_y R_2 \ldots$ $R_m <<_y T_2$ at all nodes y, so $T_1 <<_y T_2$ at all nodes y by the transitive property of the relation. If T_2 is processed first at the central node, then $T_2 <<_y T_1$ at all nodes y. As an immediate consequence, if $T_1 < T_2$ in any WLCA schedule, then $T_1 <<_y T_2$ at all nodes y. In turn, this property can be used to immediately show that all WLCA schedules are consistent. (By the way, notice that saying that $T_1 <<_y T_2$ at all nodes y is not the same as saying that all actions of T_1 precede all actions of T_2 everywhere, or $T_1 << T_2$).

Even though the WLCA has these properties, it turns out that not all schedules of update transactions and queries performed locally are consistent. The following example illustrates this. Consider two update transactions T_p and T_q in a system with two nodes x and y, and say that x is the central node. Transaction T_p simply updates item 1 without reading any data, while T_q similarly updates item 2. Thus,

$$\mathbf{T_p} = ((T_p, w, d[1,x]),(T_p, w, d[1,y]))$$

$$\mathbf{T_q} = ((T_q, w, d[1,x]),(T_q, w, d[2,y]))$$

Notice that these two transactions do not conflict because they reference different items. Thus, the name T_q will not be in WFL(T_p) and the name T_p will not be in WFL(T_q). This means that the actions of T_p and T_q may occur in any order at node y.

Next, consider two queries which read items 1 and 2. One query, Q_1, is performed locally at node x, while query Q_2 is executed at node y. We can represent Q_1 and Q_2 as

$$\mathbf{Q_1} = ((Q_1, r, d[1,x]),(Q_1, r, d[2,x])) \text{ and}$$

$$\mathbf{Q_2} = ((Q_2, r, d[1,y]),(Q_1, r, d[2,y])).$$

Now consider the following schedule of these four transactions:

$$S = ((T_p,\ w,\ d[1,x]),$$
$$(Q_1,\ r,\ d[1,x]),(Q_1,\ r,\ d[2,x]),$$
$$(T_q,\ w,\ d[2,x]),(T_q,\ w,\ d[2,y]),$$
$$(Q_2,\ r,\ d[1,y]),(Q_2,\ r,\ d[2,y]),$$
$$(T_p,\ w,\ d[1,y])\ ).$$

Since the actions of T_p and T_q may occur in any order at node y, schedule S is a legal schedule as far as the WLCA is concerned. But notice that $T_p < Q_1 < T_q < Q_2 < T_p$ (see definitions 1 and 3), so this schedule is not consistent (see theorem 1).

The fact that S is not consistent can be interpreted as follows: In S, Q_1 sees the effects of T_p but does not see the effects of T_q, while Q_2 sees the effects of T_q but not T_p. There is no way that these four transactions can be executed one at a time (i.e., serially) and produce this same effect. In any serial schedule, if Q_1 sees T_p and Q_2 follows Q_1 in the schedule, then Q_2 must also see T_p. This is not the case in S.

The example shows that the WLCA is not strong consistency insular. In other words, there is no query algorithm which can process Q_1 and Q_2 locally, without outside information, and avoid inconsistent schedules. Notice that when Q_1 is processed at node x, T_p has completed there and there is no way of knowing that T_q will come later. Hence, any query algorithm should allow Q_1 to be processed there. A similar statement can be made about Q_2. Of course, non-local query algorithms can avoid the problems. For example, if all queries are executed at the central node (just like update transactions), then all schedules will be consistent. (Such a query algorithm is not strong consistency insular because queries are not processed at their originating node, even though the items requested have values there.)

Fortunately, the fact that our sample schedule S is not consistent does not mean that Q_1 and Q_2 do not see a consistent view of the data. If we eliminate Q_2 from S, we observe that the resulting schedule is consistent and equivalent to the serial schedule T_p, Q_1, T_q (i.e., T_p executed first, then Q_1, then T_q). Thus, Q_1 sees a consistent database at node x. Similarly, if we delete Q_1 actions from S, we find that Q_2 sees the consistent data at node y produced by serial schedule T_q, Q_2, T_p. In other words, both queries see a database produced by some serial execution of the update transactions, but these serial executions may be different for each query.

How critical is it that Q_1 and Q_2 do not see "compatible" consistent views? If the users who submitted Q_1 and Q_2 communicate directly and compare their query results, then they may be confused. But on the other hand, many users may be content with getting a consistent view, especially if their weak consistency query can be processed faster than a strong consistency query. Thus, schedule S may be considered a "good" schedule in some cases, even though it is not consistent. (See section 2.)

The following theorem confirms the fact that weak consistency queries can indeed be processed locally in conjunction with the WLCA. The query algorithm for weak consistency queries can be the local locking algorithm, LLA (or any such algorithm which satisfies conditions 1 and 2 of section 4.1.1).

Theorem 4: In all schedules produced by the WLCA and LLA, queries get a consistent view of the data.

Proof: Consider any such schedule S and any query Q in that schedule. Construct schedule S' by eliminating from S all actions of queries other than Q. Since the eliminated queries had no effect on the values read by Q, query Q reads the same values in both S and S'. We will show that these values are consistent by showing that schedule S' is consistent.

Let x be the node where Q was executed, and assume that there is a cycle in the "<" relation for S'. Assume that Q is in the cycle, and say the cycle is $Q < T_1 < T_2 < \cdots < T_n < Q$.

Since $T_n < Q < T_1$, we know by conditions 1 and 2 that $T_1 \neq T_n$ and an action of T_n precedes an action of T_1 at node x. For each pair in the cycle $T_i < T_{i+1}$, we know that $T_i <<_x T_{i+1}$ (property of WCLA), so $T_1 <<_x T_2 <<_x \cdots T_n$. (Recall that $T_i <<_x T_{i+1}$ means that all actions of T_i precede all actions of T_{i+1} at node x.) This is turn means that $T_1 <<_x T_n$ (relation is transitive), which contradicts the fact that an action of T_n precedes an action of T_1 at x. If Q is not in the cycle, then $T_1 < T_2 < \cdots < T_n < T_1$ becomes $T_1 <<_x T_2 <<_x \cdots <<_x T_1$, a contradiction too. Hence, S' is consistent, and all queries read consistent data. (End of proof.)

4.1.3 An Example: The Timestamp Centralized Algorithm (TCA).

There are some update algorithms where strong or weak consistency insular queries cannot be processed locally. The Timestamp Centralized Algorithm, TCA, which we will now describe, is one of these. (This algorithm was inspired by a Majority Consensus Algorithm of [Thom79]).

The TCA is similar to the WLCA, except that a different mechanism from wait-for lists is used to execute the "perform update" messages. Let us assume that the central node has a real time clock (which is never set back). In addition to this, each item value in the system has a *timestamp* associated with it. The timestamp of value $d[i,x]$, $ts(d[i,x])$, represents the last time when item i was modified at the central site, as far as node x can tell.

When a transaction T is executed at the central node, it is assigned a timestamp, $ts(T)$, equal to the current time. All "perform update" messages for T carry a copy of $ts(T)$. This timestamp is used to detect if the execution of the "perform update T" message would overwrite values which are more current than the ones produced by T. The details are given in the following outline of the TCA.

(1)-(4): Same steps as the WLCA.

(5) Transaction T is assigned a timestamp $ts(T)$ equal to the current time at the central node.

(6) "Perform update T" messages, which include a copy of $ts(T)$, are sent out by the central node to all other nodes, giving them the new values that must be stored at each site. The central node is done with T, and goes on to process the next update transaction that is waiting in its queue. (See step 4.)

(7) When a node x receives a "perform update T" message, it performs the following for each "$d[i,x] \leftarrow$ value" in the message: If $ts(d[i,x])$ is less than $ts(T)$ then replace the value $d[i,x]$ by the value given in the message; otherwise do nothing to $d[i,x]$. No acknowledgement is sent to the central node.

An interesting property of any schedule produced by the TCA is that $T_i < T_j$ implies $ts(T_i)$ is less than $ts(T_j)$ for all update transactions T_i, T_j. (If both of the T_i, T_j actions which cause the dependency are writes, then step 7 of the TCA guarantees that $ts(T_i)$ is less than $ts(T_j)$. Otherwise, the actions must occur at the central site because all update transaction reads occur there, and $ts(T_i)$ should also be less than $ts(T_j)$.) This property can be used to show that all TCA schedules for update transactions are consistent.

Even though the TCA schedules are consistent, the database at a node can be left inconsistent "between" update transactions. To see how this could happen, consider once more the distributed database of figure 1 and transactions T_1 and T_2 of section 3.2 (T_1:"deposit 5 dollars", T_2: "withdraw 10 dollars"). Let us assume that node 1 is the central node. Then the events described in section 3.2 could occur under control of the TCA. That is, transaction T_1 could be executed at time t_1 at node 1 and message M_1 (now with timestamp t_1) could be sent to node 2. (See figure 4). Similarly, message M_2, with timestamp t_2, could also be sent to node 2.

Suppose that M_2 arrives at node 2 before M_1. The new values for the items indicated in M_2 would be stored (with timestamp t_2), leaving the database at node 2 inconsistent: d[deposits, 2] = 105, d[withdrawals, 2] = 40, d[balance, 2] = 65. (Recall that the consistency constraint is deposits − withdrawals = balance.) Any local query performed at node 2 after M_2 has been performed and before M_1 has arrived will read inconsistent data. No local query processing algorithm can avoid this because at node 2 there is no way to know whether there is or there is not a missing "perform update" message.

Of course, when message M_1 (with timestamp t_1) finally arrives at node 2, the local database will return to a consistent state. Notice that the "d[balance, 2] $\leftarrow$ 65" part of M_1 (see figure 4) will not be performed because the timestamp of M_1 is less than the timestamp of value d[balance, 2].

Thus, the TCA is neither strong nor weak consistency insular. Since the schedules of update transactions produced by the TCA are consistent, then strong and weak consistency queries could be processed as if they were update transactions. Of course, such an algorithm for queries would be non-local because queries would have to be forwarded to the central node.

4.1.4 An Example: The Distributed Locking Algorithm (DLA).

In this section we present a latest-bound insular update algorithm. Recall that a latest-bound query Q submitted at time t must see the effects of all update transactions that have committed before time t. (This algorithm was inspired by Ellis' ring algorithm [Elli77]).

In the Distributed Locking Algorithm (DLA), each node in the system has a single *update lock*. Before an update transaction T can be performed, it must request and obtain this update lock from every single node in the system. Once transaction T has all locks, it knows that no other transaction is updating, so T can be performed, releasing all locks upon its completion. Of course, global deadlocks may occur when two transactions attempt to obtain the update locks, but we will avoid this problem by assuming that there is a special program in the distributed database which detects global deadlocks and orders update transactions to release their locks.

The following is a brief description of the DLA, while figure 6 illustrates these steps.

(1) Update transaction T arrives node x from a user.

(2) Node x, on behalf of T, sends "request update lock" messages to all nodes in the system (including one to itself).

(3) When a node y receives a "request update lock" message from node x, it checks its lock. If the lock is available, then a "lock granted" message is sent to x and the lock is marked as given. If the lock has been given to some other transaction the lock request is placed on a queue to wait for the release of the lock.

(4) When node x receives "lock granted" messages from every node (including itself), it performs T. That is, the item values requested by T are read at node x, and necessary computations are carried out, and the new values are stored locally. The update lock at node x is then released.

(5) "Perform update T" messages are sent by node x to all other nodes giving them the new values that must be stored at each site.

(6) Each node y that receives a "perform update T" message stores the new values into the local database. The update lock at node y is released and any waiting lock request is granted (see step 3).

In the DLA, an update transaction must complete all of its actions at every node before the next transaction can be started. This means that transactions are executed serially, and hence, the DLA produces consistent schedules for update transactions. This also means that a query which is executed locally at node x will see the effects of all previous update transactions, except possibly the one that is currently being executed. However, by examining the local update lock at node x, it is possible to tell if indeed the query may miss the effects of the currently executing update transaction. The reason for this is that no update transaction can reach its commit point unless all update locks are held by it. (For the DLA, the commit point of an update transaction is the time of its first write when the new values first become available to queries). Thus, if the local update lock at node x is free, no update transactions are currently being executed and any local query at node x will see the effects of all previously committed update transactions.

This discussion has given us an algorithm for processing latest-bound queries locally at a node x. Say a latest-bound query Q arrives at node x at time t. Node x waits until its local update lock becomes free and at that time starts processing Q. (The local update lock does not have to be held during the processing of Q). This guarantees that Q will see the effects of all update transactions which committed before time t.

The reason why the DLA permits local latest-bound query processing is that all nodes participate in the "decision" to perform an update transaction. That is, all nodes must grant an update lock to an update transaction before it can go ahead and commit. This implies that every node must be aware of the fact that there is an update transaction in progress.

On the other hand, in may update algorithms the decision to perform an update transaction is taken by a single node or by a subset of nodes. In these cases, nodes which do not participate in the decision will be unable to process latest-bound queries locally because these nodes will not know whether an update transaction is in progress. This is the case with the CCA, WLCA and TCA algorithms. In all these algorithms, the central node decides what update transactions to process. Thus, all nodes (except the central one) must process latest-bound queries by either reading the data at the central site or by requesting a wait-for list or timestamps from the central node. (In the second case, a wait-for list for the referenced items or the current central node timestamps for the required items can be used to make sure that all necessary updates are seen by the query).

4.2 Processing Non-Insular Queries.

In the previous sections we presented several types of R insular update algorithms. In those sections we assumed that all data was completely replicated, so that all queries became insular queries. We now go back to the general case so we may discuss non-insular queries.

We have proposed a criterion for evaluating query algorithms according to the way in which they process insular queries. We would also like to have a similar criterion for evaluating query algorithms by the way they process non-insular queries. Unfortunately, in the non-insular case it is not as easy to decide what constitutes a good algorithm. That is, one non-insular algorithm may seem the best under certain circumstances, but another algorithm may be superior under different circumstances. We will illustrate this with some examples. The update algorithms we select for these examples are modified versions of the CCA, WLCA, TCA and DLA.

The CCA, WLCA, TCA and DLA as stated in the previous sections, cannot be used in a case where some nodes may not have values for all items. However, they can easily be modified to handle the more general case. The resulting algorithms may not be very practical, but they will help us to illustrate some ideas. In all algorithms, nodes should simply ignore any parts of the "perform update" messages which do not apply. For example, if a node x receives a "perform update" message indicating that the value of item i is now 10, and node x does not have a value for item i, then x should ignore that part of the message. The only other necessary modification to the algorithms is in reading data for transactions. In the CCA, when the central node decides to execute a transaction T, the values needed by T may not be available locally. Hence, the central node must send messages to other nodes requesting the values. Since the central node executes transactions serially, this modification does not change the properties of the CCA. A similar modification can be used for the other algorithms. However, care must be taken to preserve the basic properties of the algorithms. For example, in the WLCA, when the central node reads a value for item i at node x, it must ensure that node x has performed the modifications for transaction LAST(i). Otherwise, $T_i < T_j$ would not imply that $T_i <<_x T_j$.

Since the modifications to the update algorithms did not alter the basic properties of the algorithms, the modified algorithms still have the same properties with respect to insular queries. We now look at non-insular query processing with these modified algorithms. We will use the same names for the modified update algorithms as for the original algorithms, since it should be clear that we refer to the modified algorithms when discussing non-insular queries.

Consider a non-insular query Q in a system that uses an update algorithm like the TCA, which is not weak consistency insular. Suppose that Q requires values located at nodes x_1, $x_2, \ldots, x_m$. If Q reads the values at those nodes, the values obtained at each node may be inconsistent, and as a consequence, the collection of values read at nodes $x_1, \ldots, x_m$ may also be inconsistent. Thus, with the TCA, a non-insular weak (or strong) consistency queries cannot be processed by simply going to the nodes which have the required data. As in the case of insular queries, one solution is to process weak consistency queries as if they were update transactions. For the TCA, this involves processing queries at the central node. Depending on how heavily loaded the central node is, the query algorithm may or may not be efficient.

Next, consider an update algorithm which is weak consistency insular like the WLCA. Again, suppose that query Q requires values from nodes $x_1, \ldots, x_m$. In this case, it would be possible to get a consistent view at each of the m nodes, but the combination of these m consistent views may be inconsistent. Thus, even though the WLCA is weak consistency insular, non-insular weak consistency queries have the same problem as they had in the TCA.

However, in the case of the WLCA, there are two (or more) possible weak consistency algorithms. The first solution is, as for the TCA, to process non-insular queries at the central node. This algorithm could be used for weak as well as strong consistency queries. The second solution is to read a set of consistent views at the required nodes, and then to make these views consistent among themselves. As we will see, this algorithm avoids communication with the central node (unless data must be read there).

We now give a brief outline of this query algorithm which does not communicate with the central node. Let us call this algorithm the Synchronized Local Locking Algorithm (SLLA). The algorithm is presented for the case of a query which spans two nodes only. The extension for general non-insular queries should be straightforward.

(1) Query Q needs to read item values at nodes x and y.

(2) Query Q uses the LLA to obtain a consistent view of the data at node x. These values are saved in DATA(x). Before releasing the local locks, Q copies the list of performed update transactions, DONE(x), into VIEW(x). A special procedure to collect all "perform update" messages that are performed at node x in the future is started. (The reason for this will become apparent later). Then the local locks at x are released.

(3) Query Q waits at node y until all "perform update" messages for transactions in VIEW(x) have been performed at node y. (That is, until VIEW(x) is a subset of DONE(y)). Then Q uses the LLA to obtain a consistent view of the data at node y. These values are saved in DATA(y). The list DONE(y) is copied into VIEW(y) and the local locks are released.

(4) Back at node x, query Q compares the transactions seen at node y, VIEW(y), with the transactions seen at node x, VIEW(x). Query Q waits until the "perform update" messages for all transactions in VIEW(y) but not in VIEW(x) have been performed at node x. At this point, the "perform update" messages for transactions missed by Q at node x, i.e., VIEW(y) - VIEW(x), have been saved at node x. Hence, Q can simply perform the missing messages on the data read originally, DATA(x). Of course, no message for transactions not in VIEW(y) should be performed. The messages that are performed on DATA(x), are performed in the same order as they were performed at node x.

We will not prove that the SLLA gives non-insular queries a consistent view, but we will present an intuitive argument. At node y, query Q is performed with the LLA and gets a consistent view of the data. That is, Q observes the database as if the transactions in VIEW(y) had been performed serially. If we let $T_1, T_2, \ldots, T_n$ be the names of the update transaction in VIEW(y), then the effect on Q at node y is as if the serial schedule ($T_1\ T_2 \cdots T_n\ Q$) had been performed. (That is, T_1 performed completely, then T_2, and so on). At node x, Q also sees the effect of $T_1, T_2, \ldots, T_n$ but possibly in a different order. However, transactions T_i

and T_j can only be observed in a different order at nodes x and y if they have no items in common. Therefore, in the schedule observed at node x, we may switch any pair of transactions whose order does not coincide with the order in $(T_1 \cdots T_n\ Q)$. So Q observes the same consistent state at both nodes x and y, and the combined values read at nodes x and y must be consistent.

We now have the problem of deciding whether the SLLA or the central node algorithm is superior for weak consistency queries in the WLCA environment. The central node algorithm (where queries are processed like update transactions) has the disadvantage that the central node can become a bottleneck. On the other hand, the SLLA avoids this problem, but at the cost of synchronizing different consistent views. This overhead involves re-visiting nodes after reading data and delays for missing "perform update" messages (which are also possible with the other algorithm). If the central node is not heavily loaded, probably the central node algorithm would be a better choice, while the SLLA would be superior is the high load case. Also notice that the SLLA cannot process strong consistency queries, but the central node algorithm can.

To end our discussion of weak and strong consistency non-insular queries, let us briefly look at the CCA. Even though this is a strong consistency insular algorithm, non-insular queries cannot simply read the required values without any inter-node synchronization. Like with the WLCA, with the CCA there are several alternatives for processing non-insular queries. One alternative is to process the queries at the central node. Another alternative is to read several consistent views and to combine them into a single consistent view, like in the SLLA. A third alternative is to delay update processing so that a query may observe the same consistent state at several nodes. Recall that update transaction processing can be halted in the CCA by not acknowledging a "perform update" message.

Still another alternative is to use the local locking algorithm (LLA) at each node, but only releasing the locks until all values at all nodes have been read by the query. Such an algorithm guarantees that whenever a query Q sees the effects of an update transaction T_1, then (1) T_1 sees the complete effects of T_1 at any node, and (2) Q also sees the complete effects of any update transaction T_2 which preceded T_1 anywhere in the system. (These conditions are similar to those of section 4.1.1.) Hence, this algorithm, like the other three we discussed for the CCA, guarantees that all schedules of queries and update transactions are consistent. Unfortunately, this last query algorithm is prone to global deadlocks as queries and update transactions compete for the local locks. (In the LLA for insular queries, only local deadlocks could occur).

Thus, the choice of a good query algorithm for non-insular queries in the CCA is just as hard as with the WLCA. Which CCA query algorithm performs better will depend on factors such as the load at the central node, the cost of detecting and correcting and global deadlocks, and the cost of delaying update transactions.

Processing latest-bound non-insular queries is considerably simpler than processing weak or strong consistency non-insular queries. The reason for this is that a latest-bound query Q which spans data at nodes $x_1, x_2, \ldots, x_n$, can be decomposed into a set of n independent insular queries at nodes $x_1, x_2, \ldots, x_n$. That is, as long as each of the sub-queries observes all update transactions which committed before Q's submission time, then the combined data produced by the sub-queries will reflect all update transactions which committed before Q's submission time this means that the algorithm for processing non-insular latest-bound queries can simply be a collection of calls to the latest-bound insular algorithm.

4.3 Processing t-vintage and t-bound Queries.

In this section we will discuss the processing of t-vintage and t-bound queries. We start by pointing out that if any type of t-vintage or t-bound queries are to be processed, then a history of the distributed database must somehow be kept by the system. This history must record the various states the distributed database went through to arrive at the current state.

In the algorithms we have presented so far, the "old" values of an item were never saved before replacing them with new values. Thus, it is impossible to find out what the distributed database looked like at a given time or at what times the previous update transactions committed. No *t*-bound or *t*-vintage queries can be performed with these algorithms as stated.

In the update algorithms that do keep a history, there are several alternatives for doing so. Probably the simplest and most efficient alternative would be to keep a *log* of all the update transactions. A transaction log is a record of all the transactions and the modifications they performed. Typical entries for update transaction *T* in the log would be *T*'s commit time, *T*'s name, the new item values produced by *T*, and the old values they replaced. A *t*-vintage or *t*-bound query can be processed by examining the log in reverse chronological order until the desired data is reconstructed. Since these queries must examine the log, *t*-vintage and *t*-bound queries can only be executed at nodes which have a copy of the log. One important advantage of the transaction log mechanism is that in many systems the log is required anyway for crash recovery and update transaction undoing [Gray77]. Thus, in these systems, keeping the log for *t*-vintage and *t*-bound queries represents no real overhead.

A second alternative for keeping a history of the distributed database is to keep a log for each item value in the system. Each of these logs records the previous values of an item at a node and the times when the values changed [Reed78]. A *t*-vintage of *t*-bound query can be processed by examining the logs for all the items referenced by the query. This can be done at a single node as long as all the required logs are found there.

Due to space limitations, we will not discuss here how either type of log can be maintained and how the *t*-vintage and *t*-bound queries can be processed using the logs.

5. CONCLUSIONS.

In this paper we studied the various requirements that read-only transactions or queries can make. The requirements can be strong consistency, weak consistency, *t*-vintage, *t*-bound or latest-bound. We also discussed how queries can be processed efficiently in a distributed database. The criterion of *R* insularity, that is, whether a query with a requirement *R* can be processed locally, was used to characterize both query and update transaction algorithms.

Several update and query algorithms were presented as examples. These examples were intended to illustrate the issues involved in query processing, and care must be taken not to over generalize. For example, one cannot say that only update algorithms which produce serial schedules (like the CCA) can be strong consistency insular. Similarly, we cannot conclude that any update algorithm which uses timestamps will not be weak consistency insular. The *R* insularity property is a property of the update and query algorithms, and not of a particular approach.

In any distributed database where queries are expected to be a significant fraction of the total transactions, the expected requirements of the queries should be studied and considered in the design of the update and query algorithms. The *R* insularity properties of the proposed algorithms should be analyzed to see if the algorithms are appropriate. For most applications, it is not a good idea to design the most efficient or most elegant update algorithm without keeping in mind queries and their requirements.

It is also important to notice that in some cases a minor change of the update algorithm can drastically change its *R* insularity properties. For example, in both the TCA and the WLCA algorithms, the central node can add a sequence number to all update transactions processed. The sequence numbers can then be appended to the "perform update" messages, and the query algorithms can use this extra information to process weak consistency and even strong consistency queries. Thus, the modified TCA and WLCA become strong consistency insular algorithms. But of course, in algorithms where there is no central node to assign sequence numbers, it may not be as easy to obtain the necessary information to change the *R* insularity properties.

In closing, let us point out that the ideas presented in this paper could also be applied to centralized databases. In a centralized database queries can have the same requirements that we have discussed here. However, in a centralized database system (when all queries are insular) it is fairly easy to design *R* insular query algorithms for any requirement *R* because all necessary information is located in the one node.

6. REFERENCES.

[Alsb76]P.A. Alsberg and J. D. Day, "A Principle for Resilient Sharing of Distributed Resources", 2nd Intl. Conf. on Software Engineering, San Francisco (1976) 562-570.

[Bern78]P. A. Bernstein, J. B. Rothnie, N. Goodman and C. A. Papadimitriou, "The Concurrency Control Mechanism of SDD-1: A System for Distributed Databases (The Fully Redundant Case)", IEEE Trans. on Software Engineering, Vol. 4, Num. 3 (1978) 154-168.

[Elli77]C. A. Ellis, "Consistency and Correctness of Duplicate Database Systems", 6th Symposium on Operating System Principles (1977) 67-84.

[Eswa76]K. P. Eswaran, J. N. Gray, R. A. Lorie and I. L. Traiger, "The Notions of Consistency and Predicate Locks in a Database System", Communications of the ACM, Vol. 19, Num. 11 (1976) 624-633.

[Garc79]H. Garcia-Molina, "Performance of Update Algorithms for Replicated Data in a Distributed Database", Report STAN-CS-79-744, Department of Computer Science, Stanford University (1979).

[Gray76]J. N. Gray, R. A. Lorie, G. R. Putzolu and I. L. Traiger, "Granularity of Locks and Degrees of Consistency in a Shared Database", Modeling in Database Management Systems, North Holland (1976) 365-394.

[Gray77]J. N. Gray, "Notes on Database Operating Systems", Advanced Course on Operating Systems, Technical University Munich (1977).

[Lamp78]L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System", Communications of the ACM, Vol. 21, Num. 7 (1978) 558-564.

[Reed78]D. P. Reed, "Naming and Synchronization in a Decentralized Computer System", M.I.T. Department of Electrical Engineering and Computer Science, PhD Thesis (1978).

[Thom79]R. H. Thomas, "A Majority Consensus Approach to Concurrency Control", ACM Trans. on Database Systems, Vol. 4, Num. 2 (1979) 180-209.

FIGURE 1

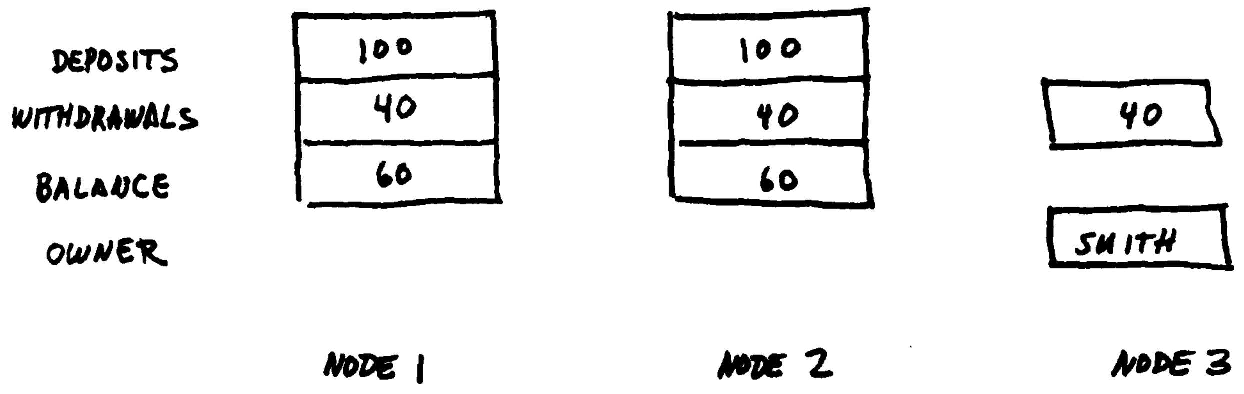


FIGURE 2

T_1 : " DEPOSIT 5 DOLLARS "

LINE	INFORMAL	FORMAL
1	$x \leftarrow d[deposits, 1]$	$\underline{T_1} = ((T_1, r, d[deposits, 1]),$
2	$y \leftarrow d[balance, 1]$	$(T_1, r, d[balance, 1]),$
3	$d[deposits, 1] \leftarrow x+5$	$(T_1, w, d[deposits, 1]),$
4	$d[balance, 1] \leftarrow y+5$	$(T_1, w, d[balance, 1]),$
5	$d[balance, 2] \leftarrow y+5$	$(T_1, w, d[balance, 2]),$
6	$d[deposits, 2] \leftarrow x+5$	$(T_1, w, d[deposits, 2])).$

FIGURE 3

T_1 : "DEPOSIT 5 DOLLARS" T_2 : "WITHDRAW 10 DOLLARS"

LINE	INFORMAL	FORMAL
1	T_1: $x \leftarrow d$[DEPOSITS, 1]	$\underline{S}$ = ((T_1, r, d[DEPOSITS, 1]),
2	T_1: $y \leftarrow d$[BALANCE, 1]	(T_1, r, d[BALANCE, 1]),
3	T_1: d[DEPOSITS, 1] $\leftarrow x+5$	(T_1, w, d[DEPOSITS, 1]),
4	T_1: d[BALANCE, 1] $\leftarrow y+5$	(T_1, w, d[BALANCE, 1]),
5	T_2: $z \leftarrow d$[WITHDRAWALS, 1]	(T_2, r, d[WITHDRAWALS, 1]),
6	T_2: $w \leftarrow d$[BALANCE, 1]	(T_2, r, d[BALANCE, 1]),
7	T_2: d[WITHDRAWALS, 1] $\leftarrow z+10$	(T_2, w, d[WITHDRAWALS, 1]),
8	T_2: d[BALANCE, 1] $\leftarrow w-10$	(T_2, w, d[BALANCE, 1]),
9	T_2: d[WITHDRAWALS, 2] $\leftarrow z+10$	(T_2, w, d[WITHDRAWALS, 2]),
10	T_2: d[BALANCE, 2] $\leftarrow w-10$	(T_2, w, d(BALANCE, 2]),
11	T_1: d[BALANCE, 2] $\leftarrow y+5$	(T_1, w, d[BALANCE, 2]),
12	T_1: d[DEPOSITS, 2] $\leftarrow x+5$	(T_1, w, d[DEPOSITS, 2]),
13	T_2: d[WITHDRAWALS, 3] $\leftarrow z+10$	(T_2, w, d[WITHDRAWALS, 3])).

FIGURE 4

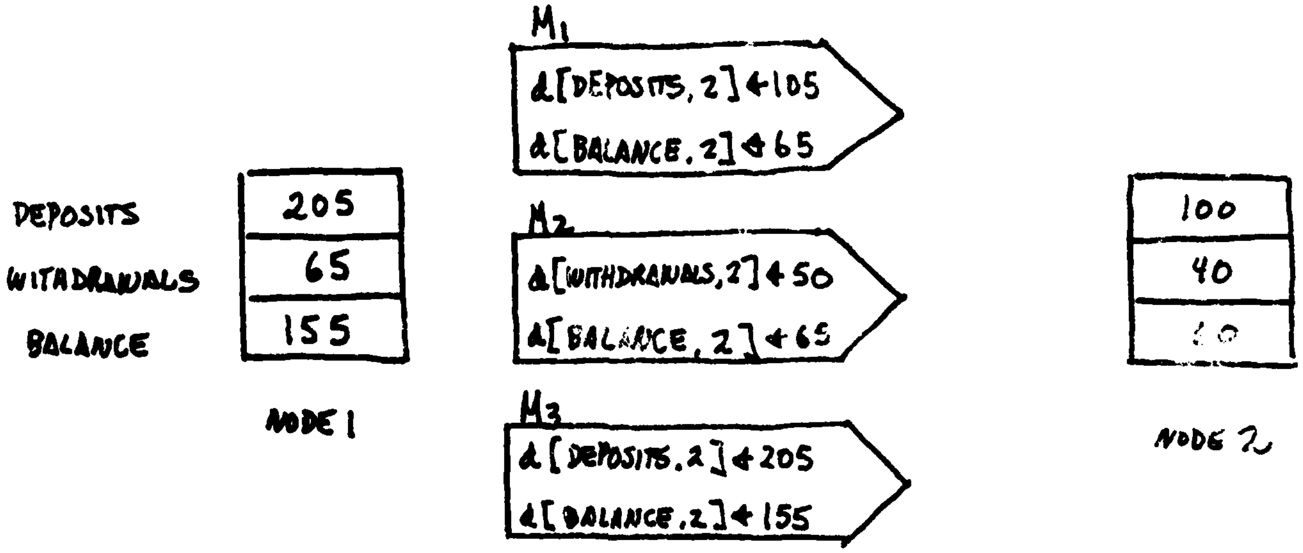
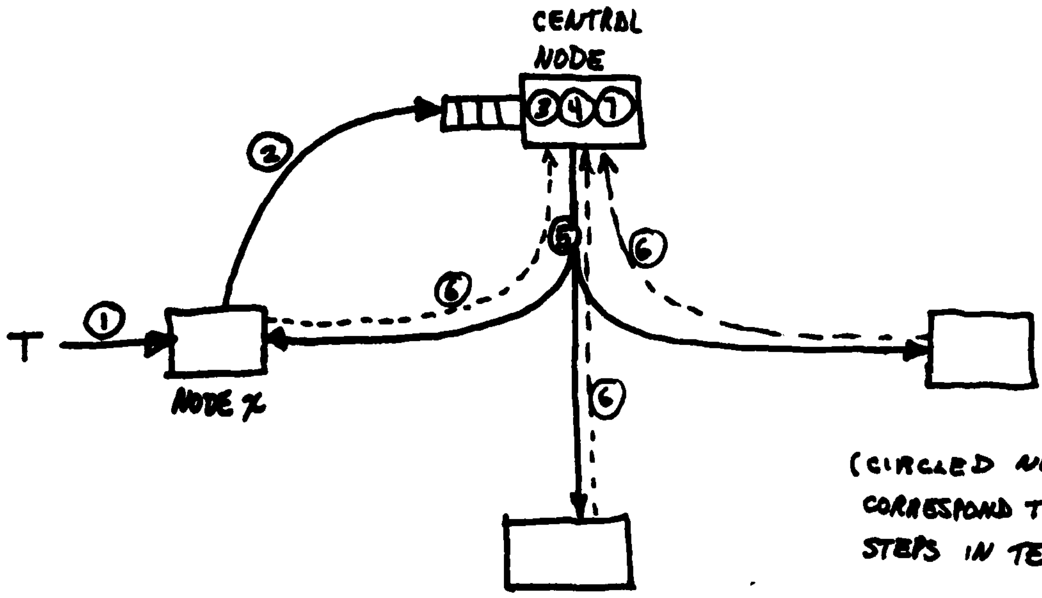


FIGURE 5

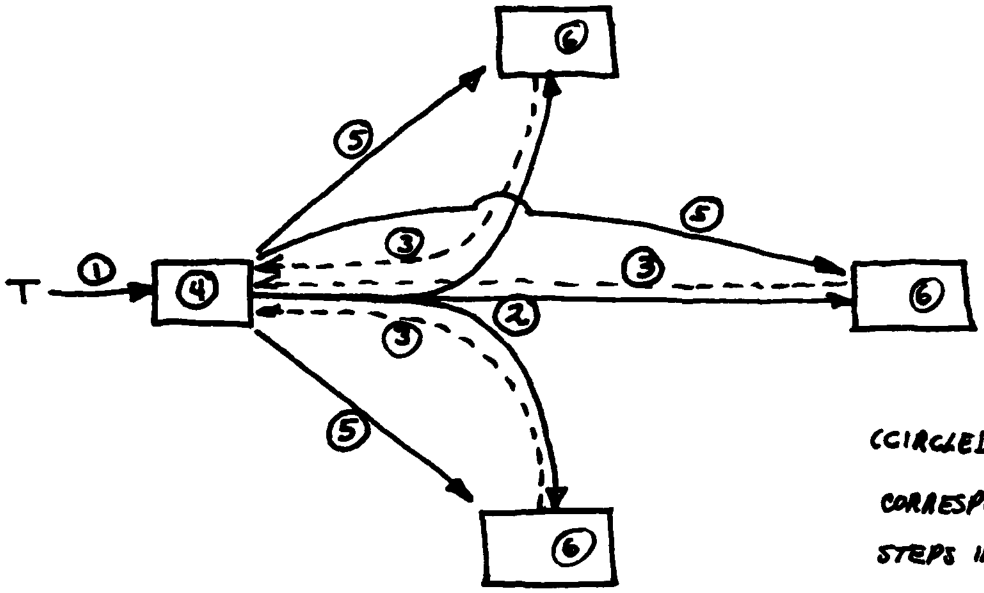
COMPLETE CENTRALIZATION ALGORITHM (CCA)



(CIRCLED NUMBERS CORRESPOND TO ALGORITHM STEPS IN TEXT.)

FIGURE 6

DISTRIBUTED LOCKING ALGORITHM (DLA)



(CIRCLED NUMBERS CORRESPOND TO ALGORITHM STEPS IN TEXT.)